**Introduction to Computer Systems: Assignment Two**

# **TASK 1 – Stop-and-Wait ARQ Implementation**

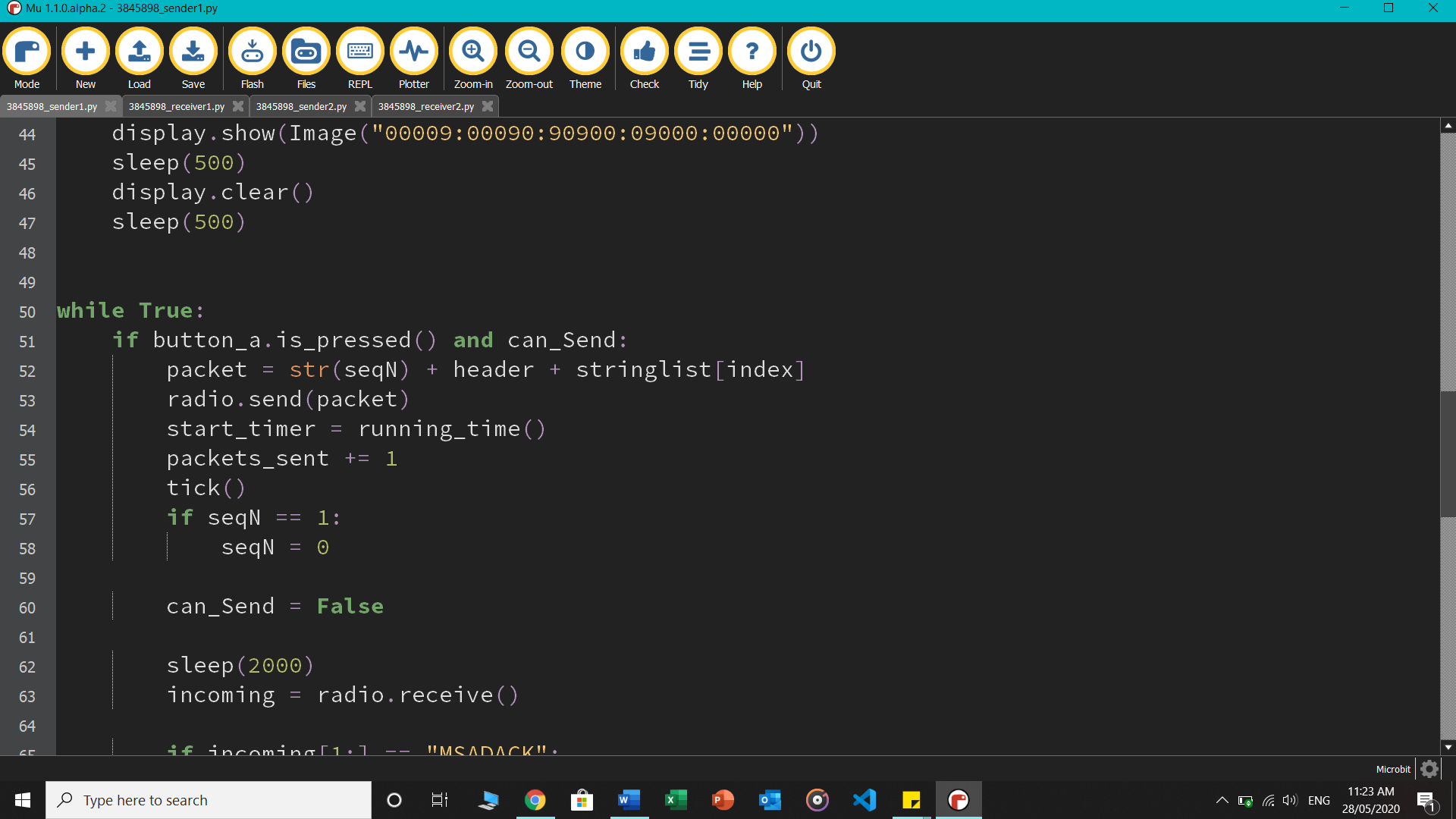
## **Design of the packets**

The design was chosen to be as simple as possible. The construction of a packet mainly involved two components: the header and the message. The header consisted of the sender’s address = “AD”, and the receiver’s address = “MS”. When tied together with the sequence number, the header and the ‘message’ looked like: “0ADMSmessage”. Hence, whenever the receiver would intercept any messages it would know that index 0 is the sequence number, followed by the sender and receiver address, while the rest is to be interpreted as the message.

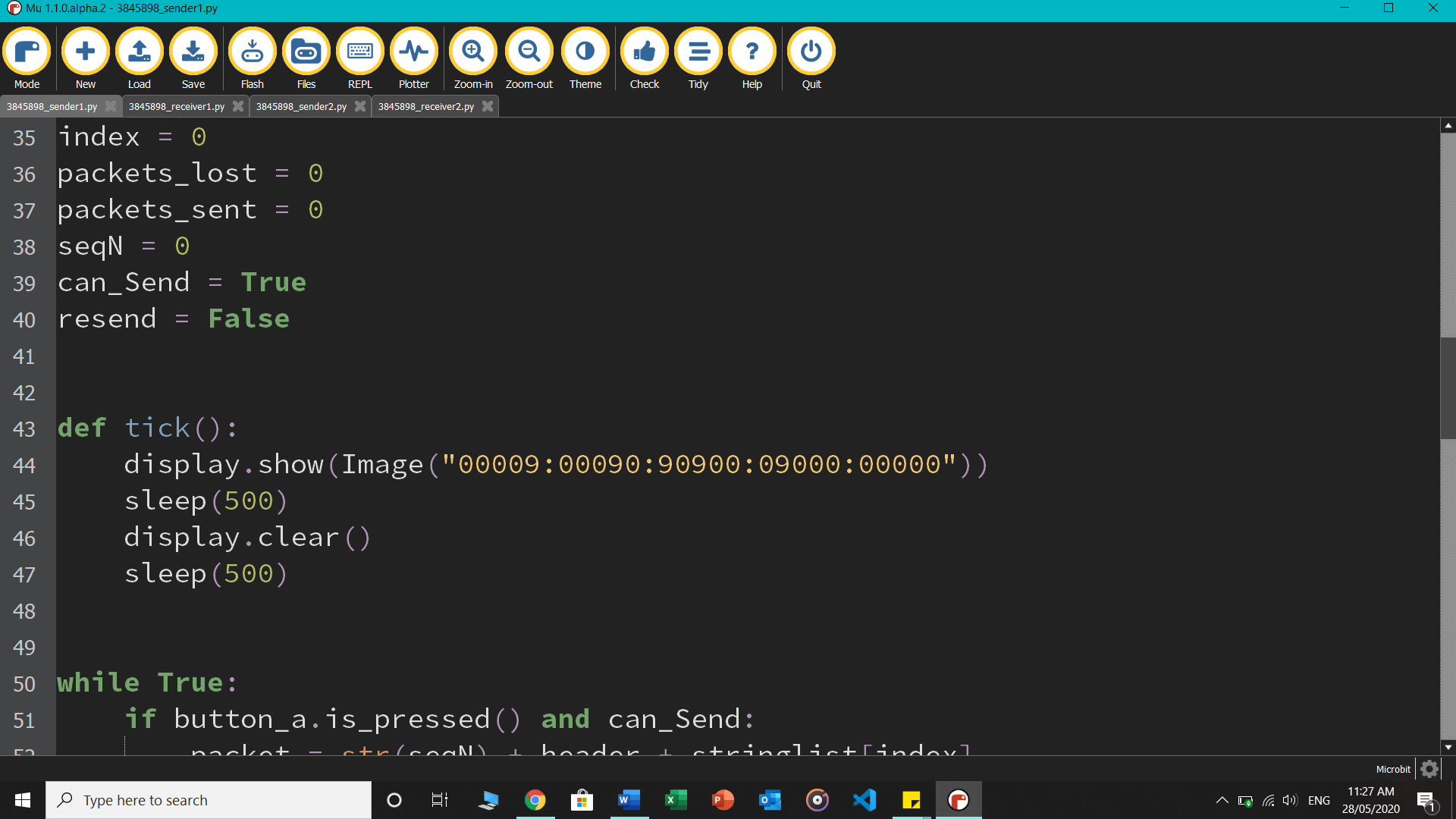
|  |  |  |  |
| --- | --- | --- | --- |
| Sequence Number | Sender Address | Receiver Address | Data [message or ack] |

**Codes used to designate various states or action**

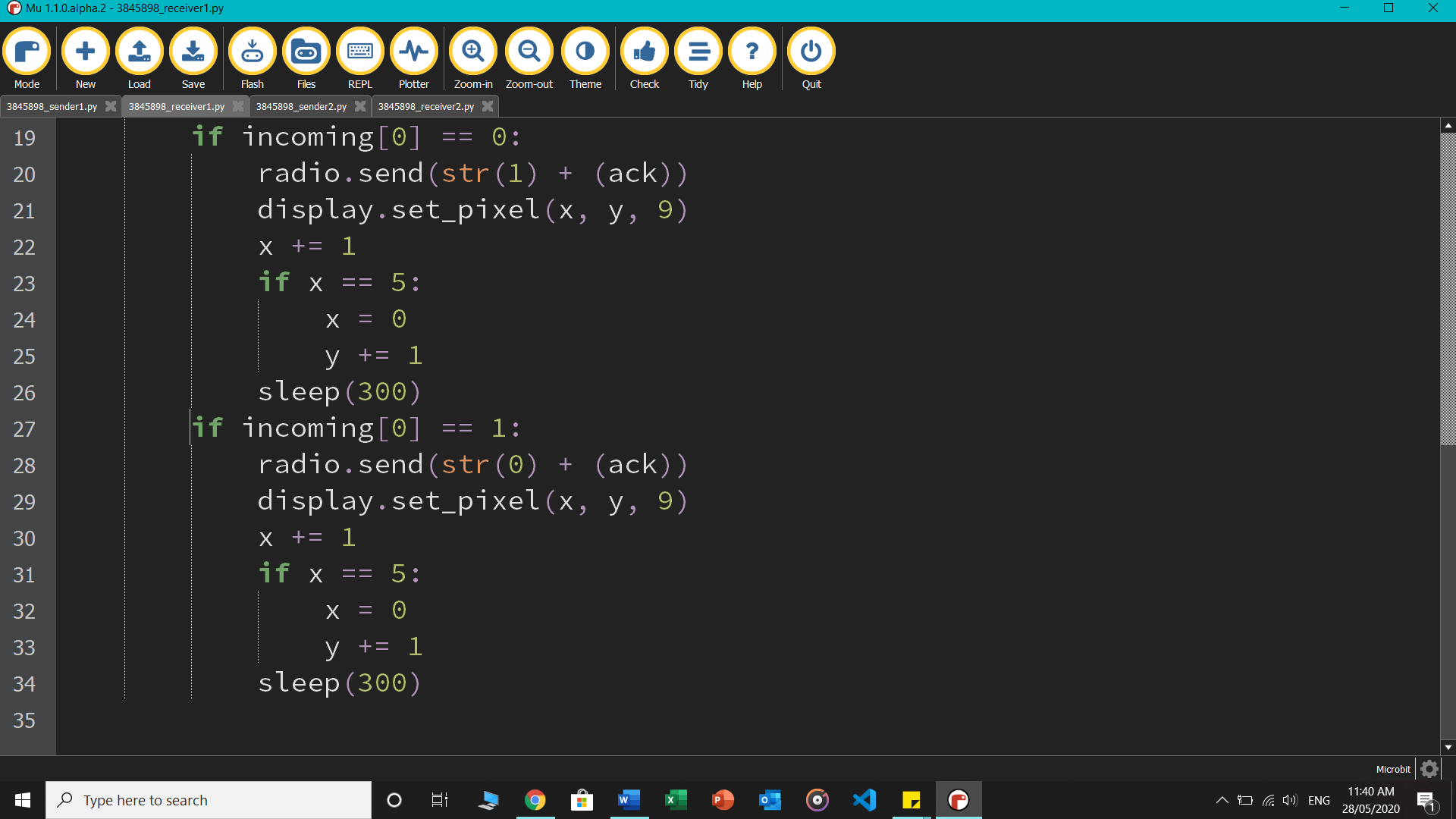
* A list made of 20 strings, each representing a frame, was sent from Node 1 to 2
* [SENDER] Packet is constructed using the given blueprint above, then it is sent, after which the *sent\_counter* increments by 1 and the timer also starts. A Boolean flag *canSend()* becomes false because now the sender is waiting for the ACK



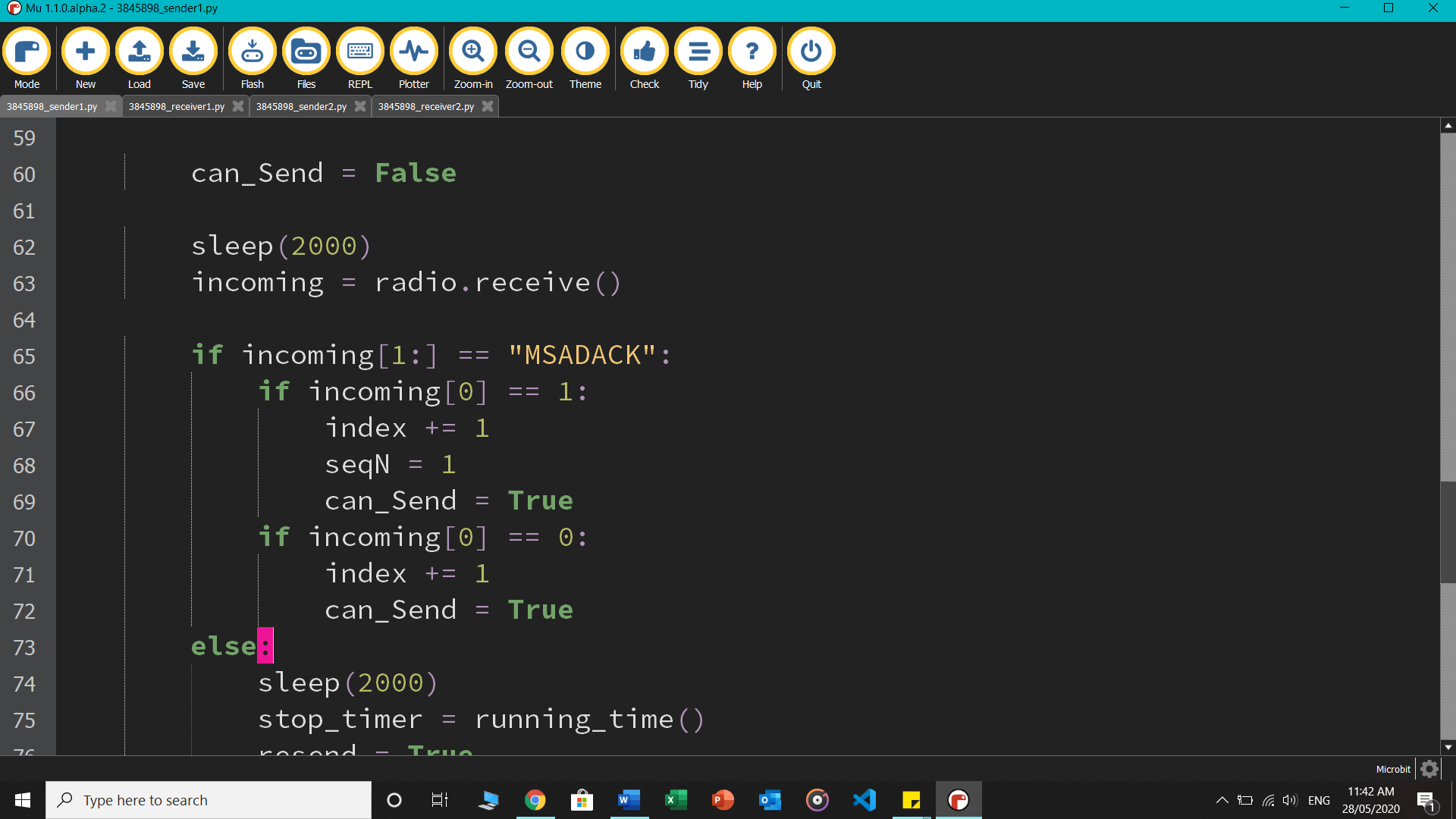
* A *tick()* function is used to display a signal on the sender that the packet was sent through



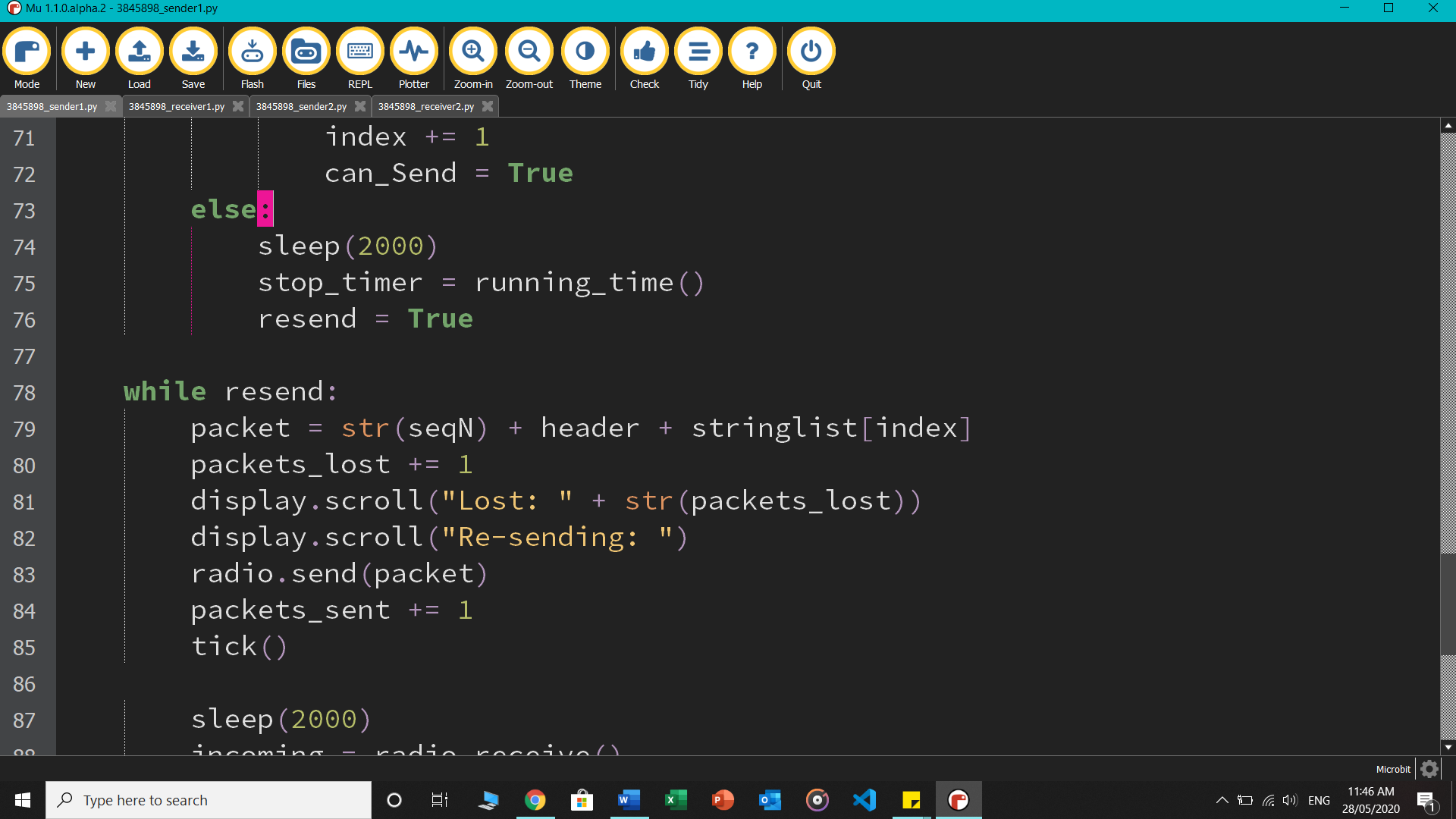
* [RECEIVER] Any message coming into the receiver is stored in the variable *incoming*. If frame 0 is received, ACK 1 is sent to the sender. And when it gets frame 1, ACK 0 is sent. Also, using *display.set\_pixel()* the status of the total frames received is showed using the LEDs on the microbit, example the top left corner LED glows when the first frame is received.



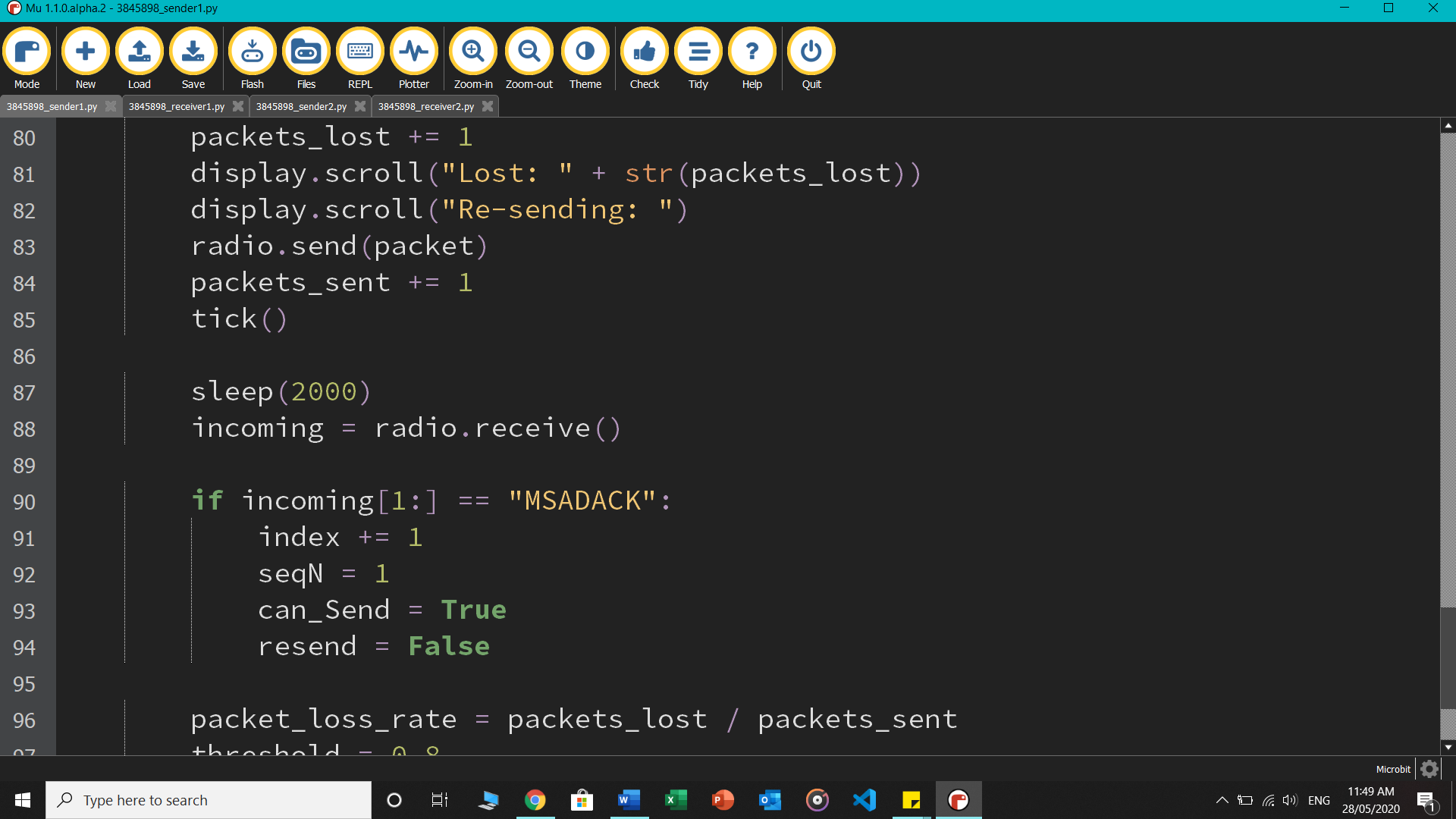
* [SENDER] If ACK is received with sequence number 1, the index in the frame list moves up and the packet will now attach 1 as its sequence number. If ACK 0 is received, frame attaches 0 in front. The flag becomes true.



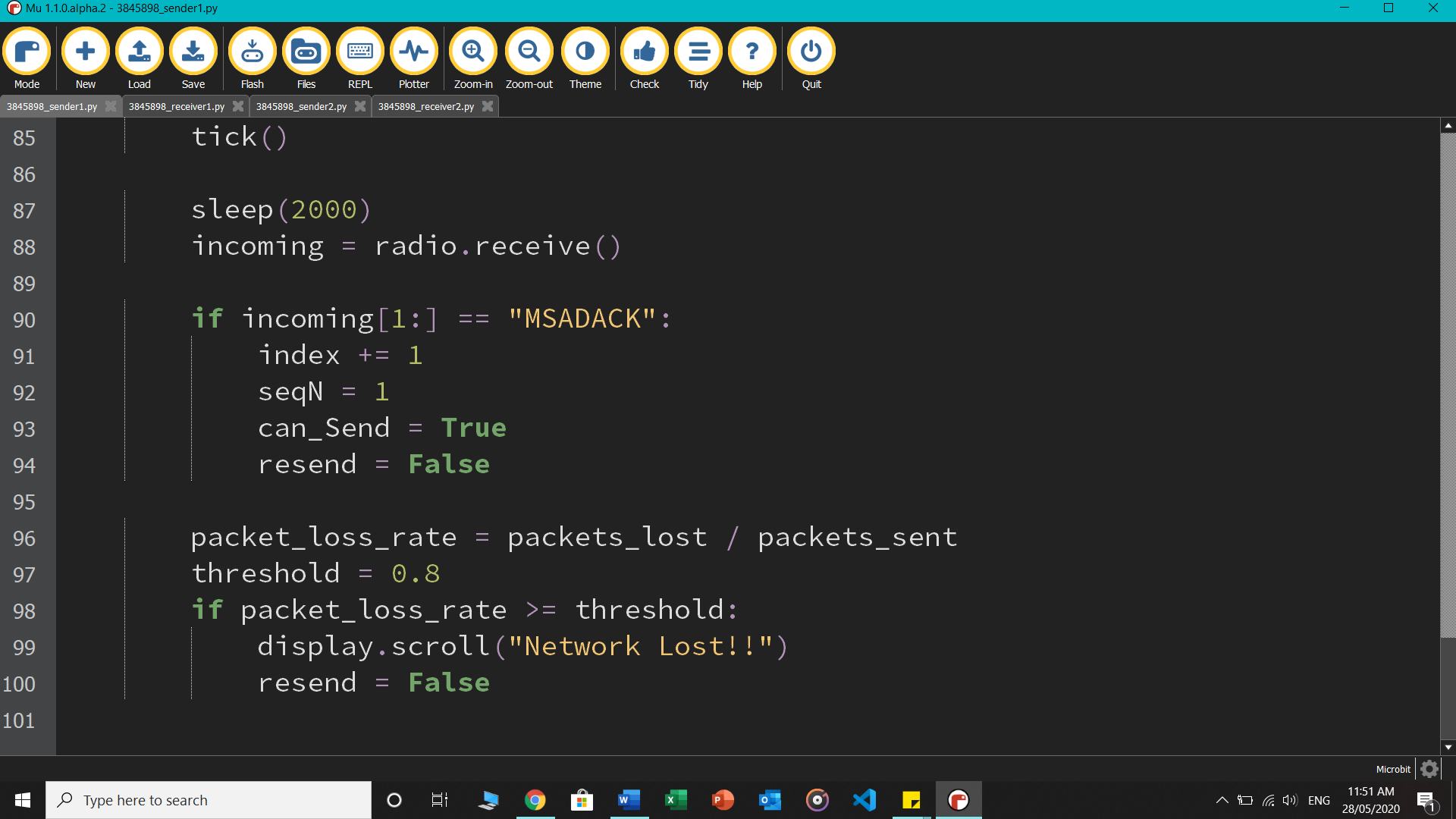
* [SENDER] If no ACK comes through, the sender times out and *resend* loop activates. Packet is re-constructed with the recent sequence number and sent. Lost\_counter increments by 1 and total packets lost statistics are scrolled on the microbit.



* After this the sender checks for the ACK again. If none turns up, it continues looping through and resending the recent frame. If ACK is received, resend loop is exited.



* In the same resend loop, packet loss rate is determined every time a resend happens. If the threshold, which is 80% is crossed, then the “Network Lost” message is scrolled and canSend() never becomes true, as the network is declared no longer usable by the protocol.



**Rules of communication**

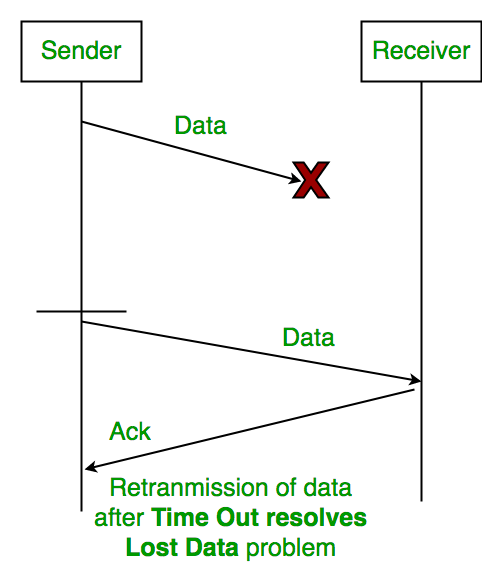
Sender sends frame 0 [meaning with sequence number 0] => receiver gets it and sends back ACK 1 => sender gets it and now sends frame 1 => receiver intercepts frame 1 and sends ACK 0, and cycle repeats for all packets.

How my system deals with errors:

In Stop-and-wait Automatic Repeat Request, three problems could occur:

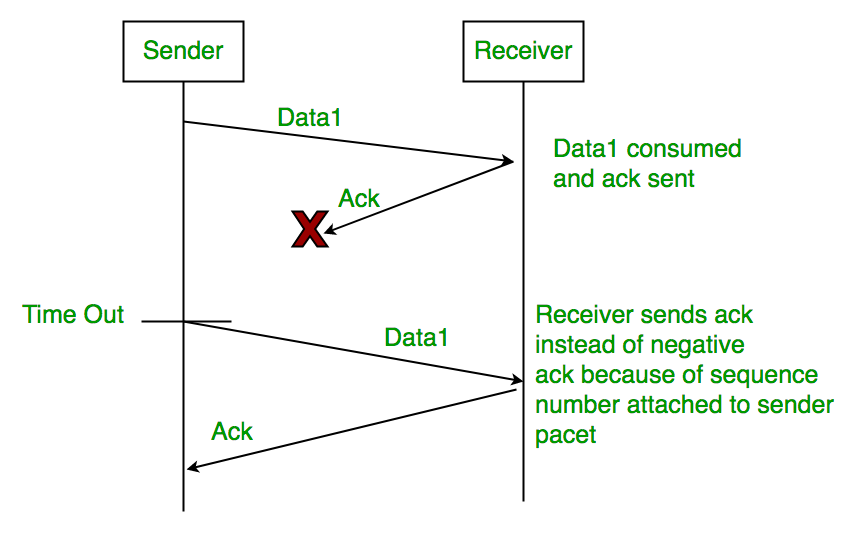
1. Lost data from the sender – the sender sends data, which gets lost in the passage and the receiver never gets it, thus it never sends ACK and the sender waits for an infinite amount of time.

Solution: Using a timeout period after which sender automatically retransmits the last frame.



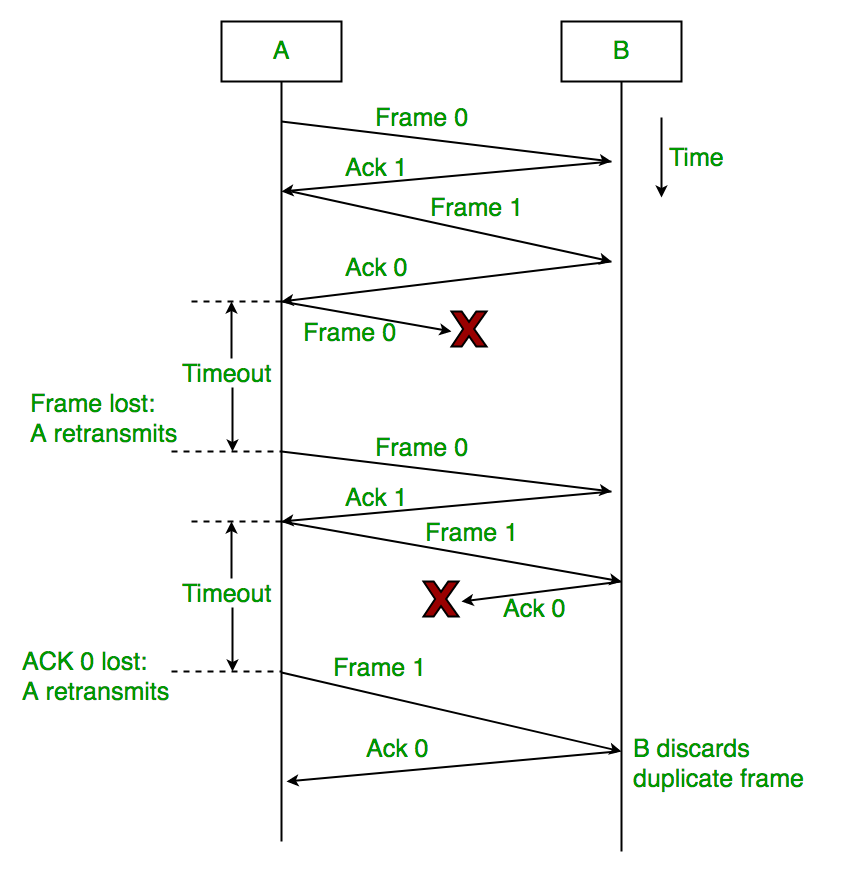
1. Lost acknowledgement from the receiver – the receiver successfully gets the data and sends off an ACK, but it gets lost in the passage and the sender never gets it.

Solution: This too can be resolved by a similar timeout period, but with the addition of a sequence number.



1. Delayed Acknowledgement/Data – after timeout on sender side, a long delayed acknowledgement might be wrongly considered as an ACK of some other recent packet.

Solution: Assigning sequence number to the frame and the ACK. If receiver doesn’t get frame 0 and instead gets frame 1, it will send back ACK 0, not 1, asking for resending of frame 0 as it was lost. Moreover, if sender has sent frame 0 and no ACK has arrived, the system will timeout and it will resend frame 0.



# **TASK 2 – Go-Back-N Implementation**

## **Design of the packets**

The structure of the packets was kept exactly similar to the first task.

[SENDER]

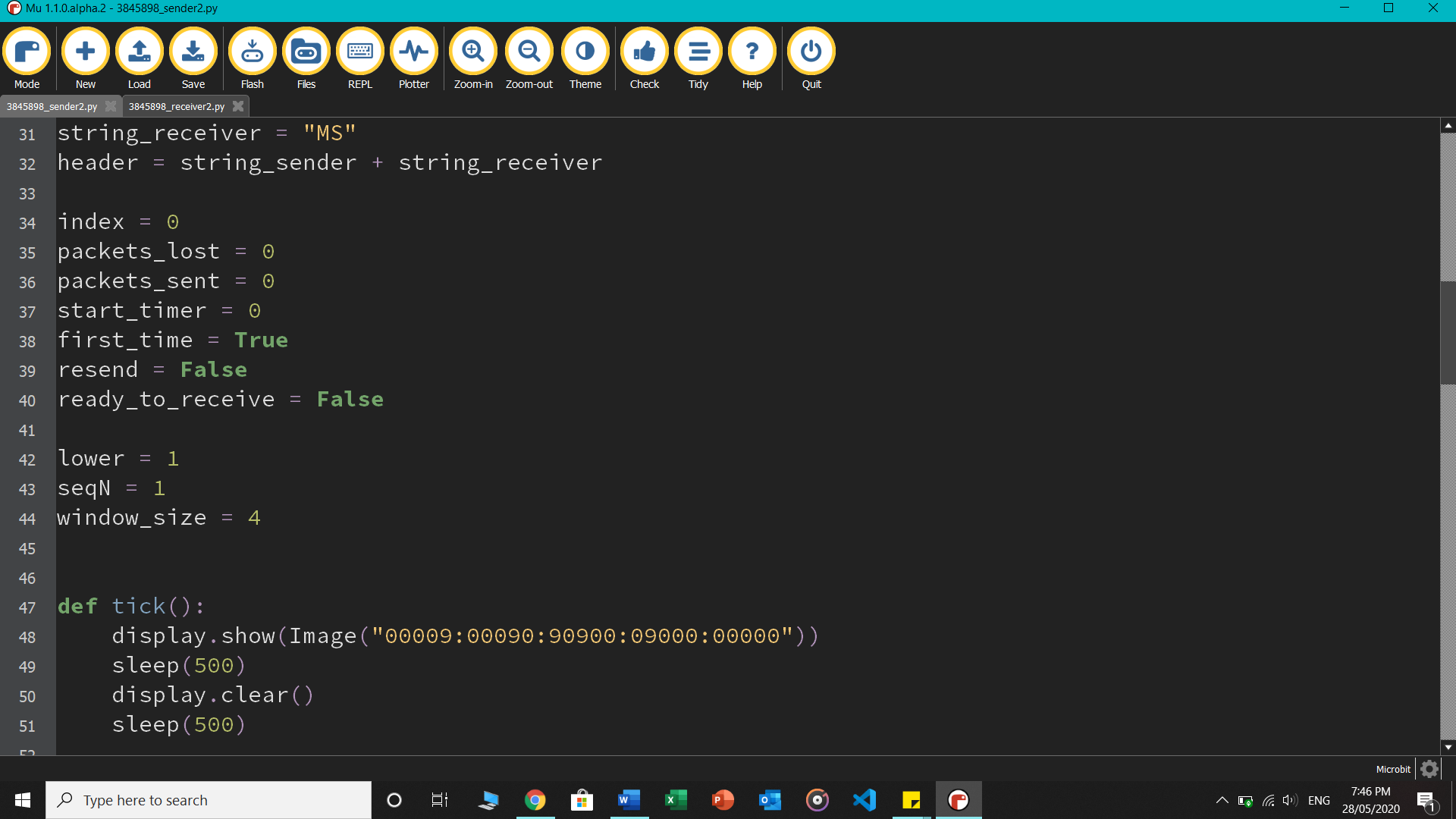
|  |  |  |  |
| --- | --- | --- | --- |
| Sequence Number | Sender Address | Receiver Address | Data [message] |

[RECEIVER]

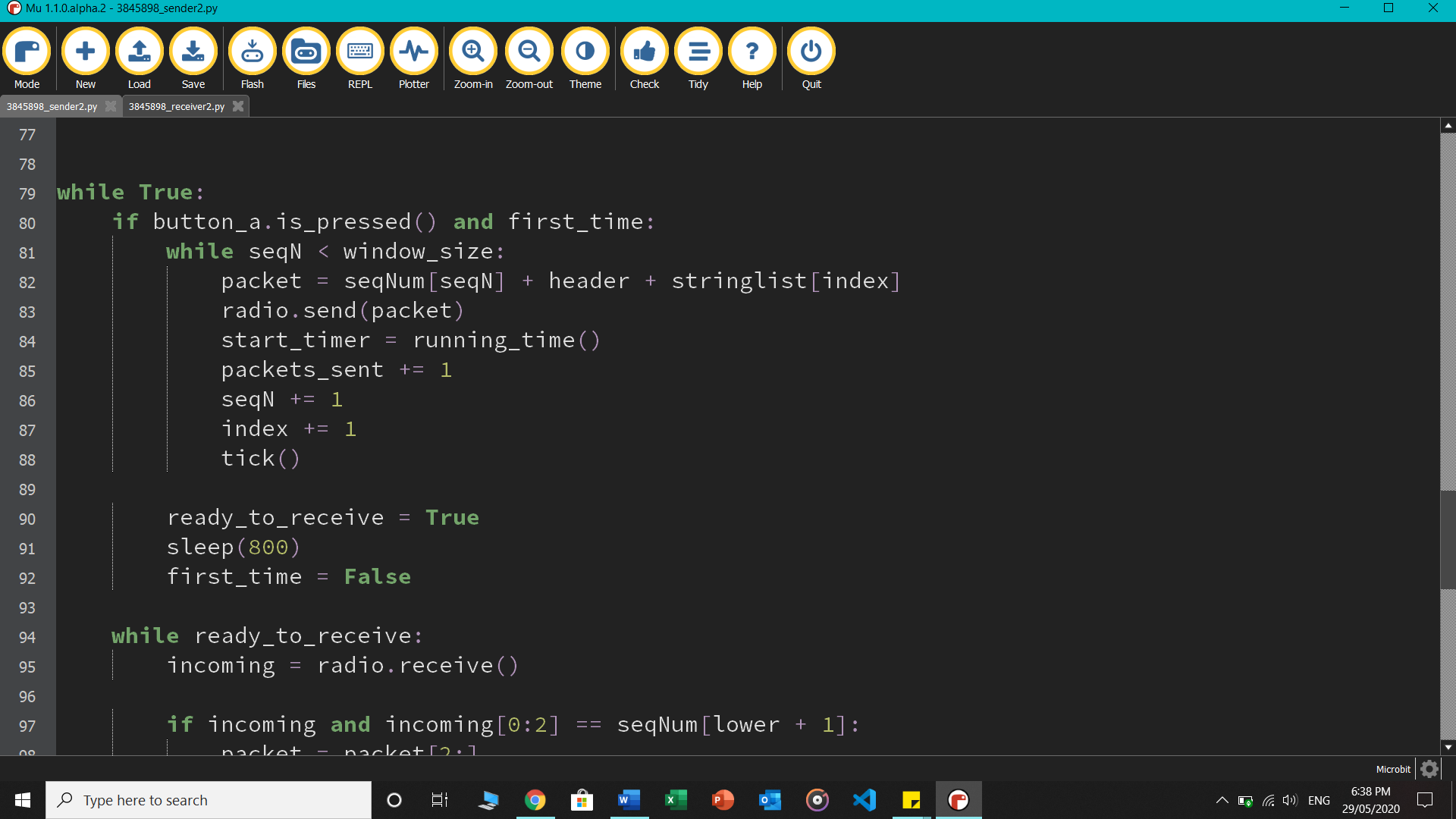
|  |
| --- |
| Sequence Number [representing the acknowledgement] |

**Codes used to designate various states or action**

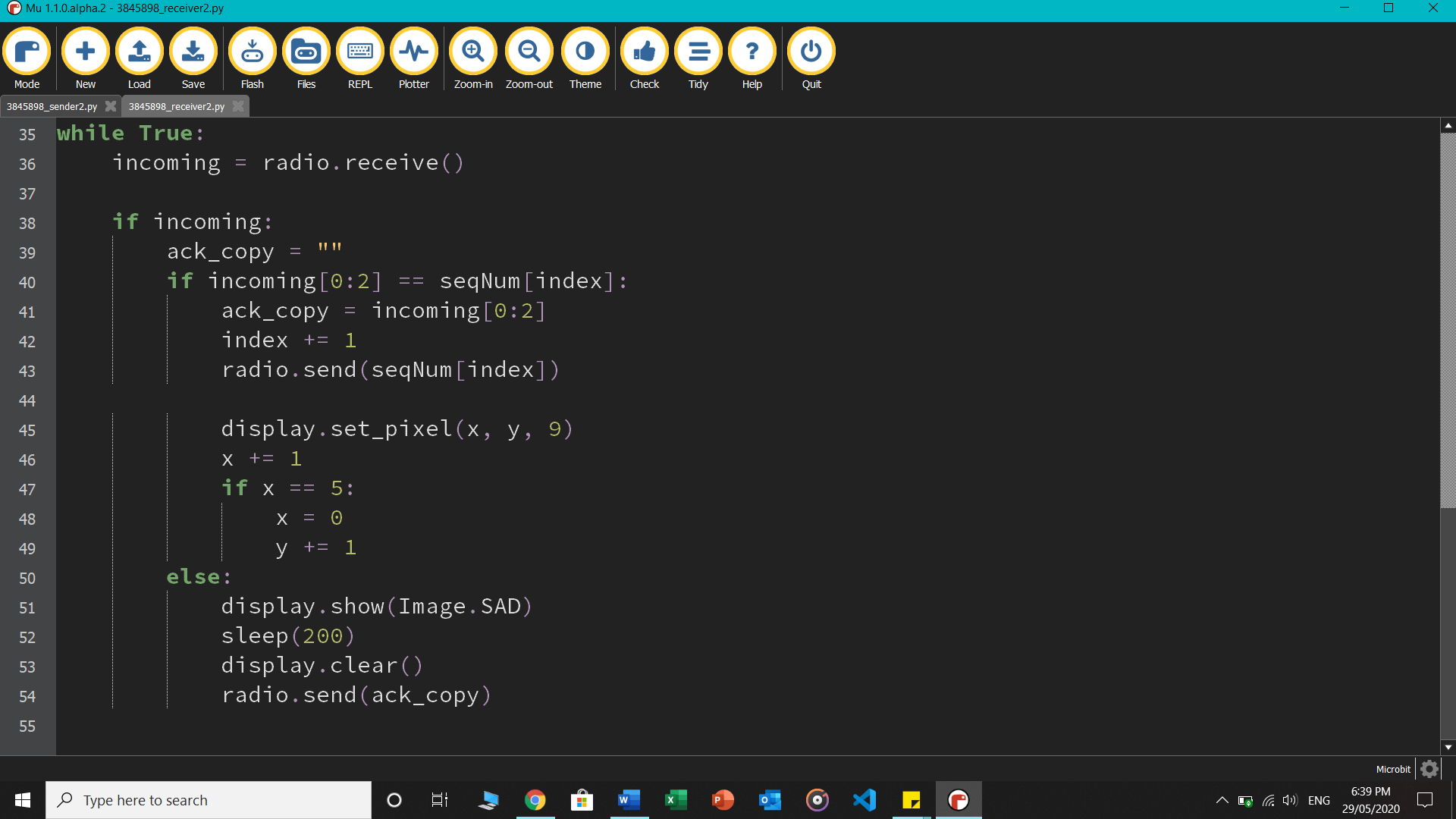
* A list made of 20 strings, each representing a frame, was sent from Node 1 to 2
* [SENDER] Following are defined: lower bound of window = 1; sequence number = 1; sliding window size = 4; and *first time* representing a simple flag that will turn off after the full window of 4 frames sent for the first time.



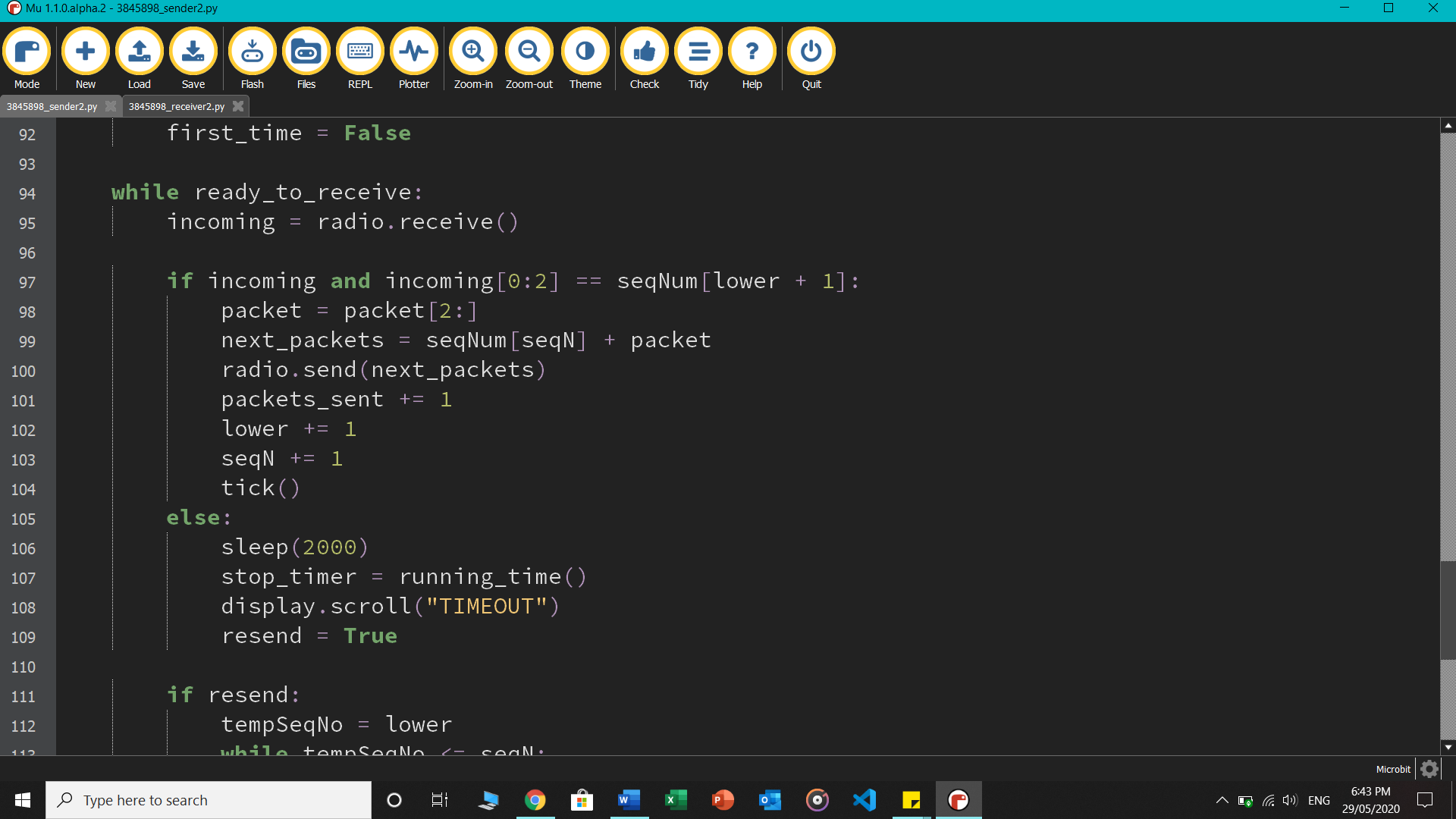
* Packet is constructed using the given blueprint above with the sequence number in front, then it is sent, after which the *sent\_counter* increments by 1 and the timer also starts. A Boolean flag *first\_time* becomes false, because now after sending a full window of 4 frames, the sender waits for the ACK and *ready\_to\_receive* loop activates. Also, *seqN* increments by 1 every time and attaches in front of the packet. A *tick()* is used to signal a successful transmission from the sender.



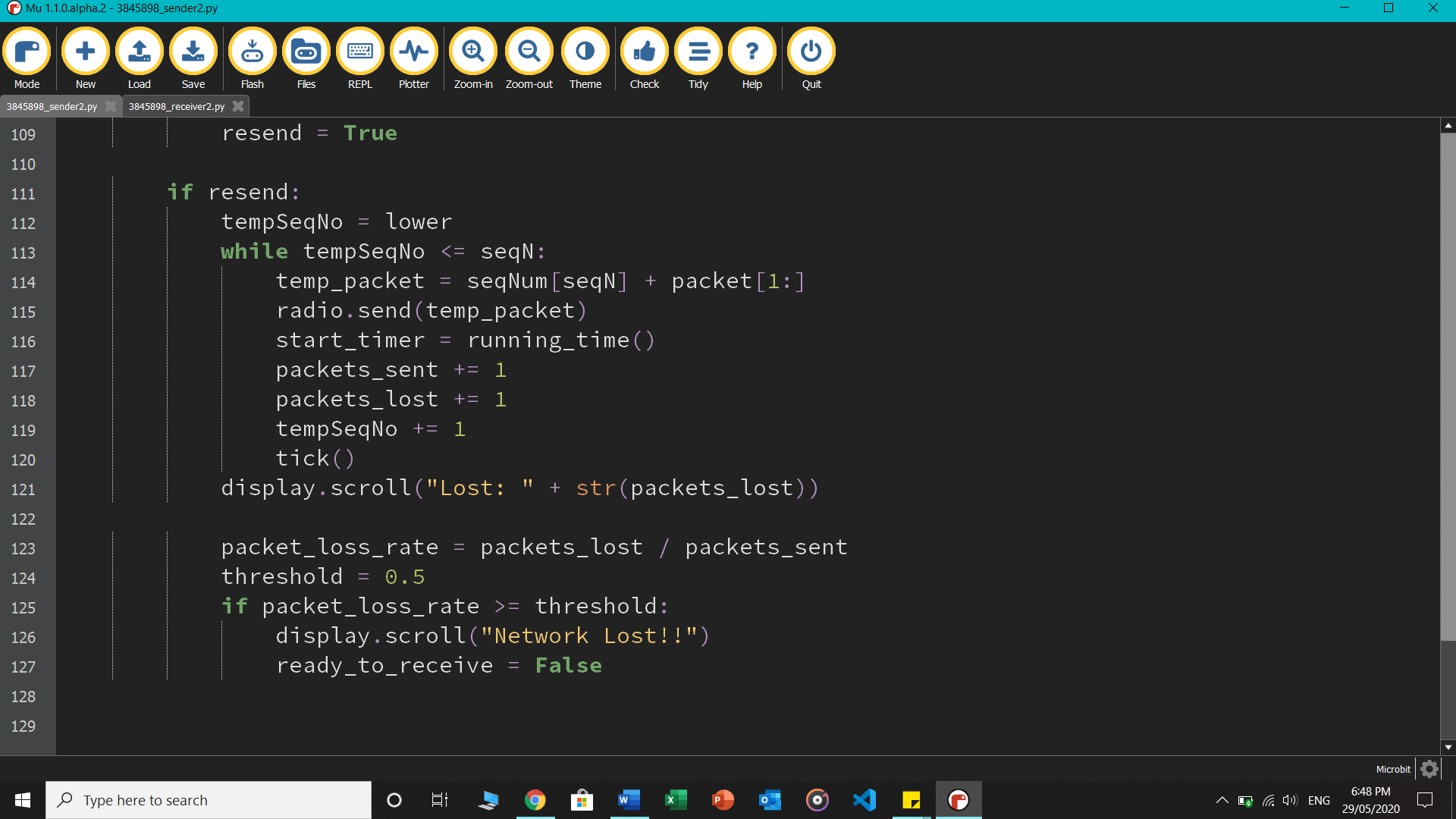
* [RECEIVER] The receiver has a pre-defined set of sequence numbers in a list, hence it always knows what sequence number it is expecting. If incoming[0:2] equals the expected sequence number, meaning the first index of the message string matches it, it turns to expect the next sequence number and sends it to the sender microbit. Also, the *display.set\_pixel()* function is used to the show the status of the transmissions, example, the first successful packet will light up the top left corner LED



* [SENDER] The ready\_to\_receive loop waits for any incoming ACK. If one comes, it matches it to the lower bound of the window (which is set up using lower as an index in the defined list of sequence numbers). If a match happens, the next frame is constructed, sent through, and the window now slides one step (depicted using lower and seqN). After every successful transmission comes a tick



* If no ACK arrives, as in a situation when the packet was lost, or the ACK was lost, timeout happens after waiting two more seconds, and resend condition becomes true
* What the following if statement does is it assigns a temporary sequence number to the last frame that wasn’t yet acknowledged, which the lower bound of the window. Starting from this up to and including seqN, all frames are then re-transmitted with the assignment of the new sequence number. Every single time, *lost\_counter* increments by 1 and a tick is displayed. After the frames are resent, the lost statistics are displayed, and the threshold value is examined against the packet loss rate using the given formula. If it is above 50%, then the network is declared as no longer usable and the loop ends.



# **TASK 3 – Error Checking**

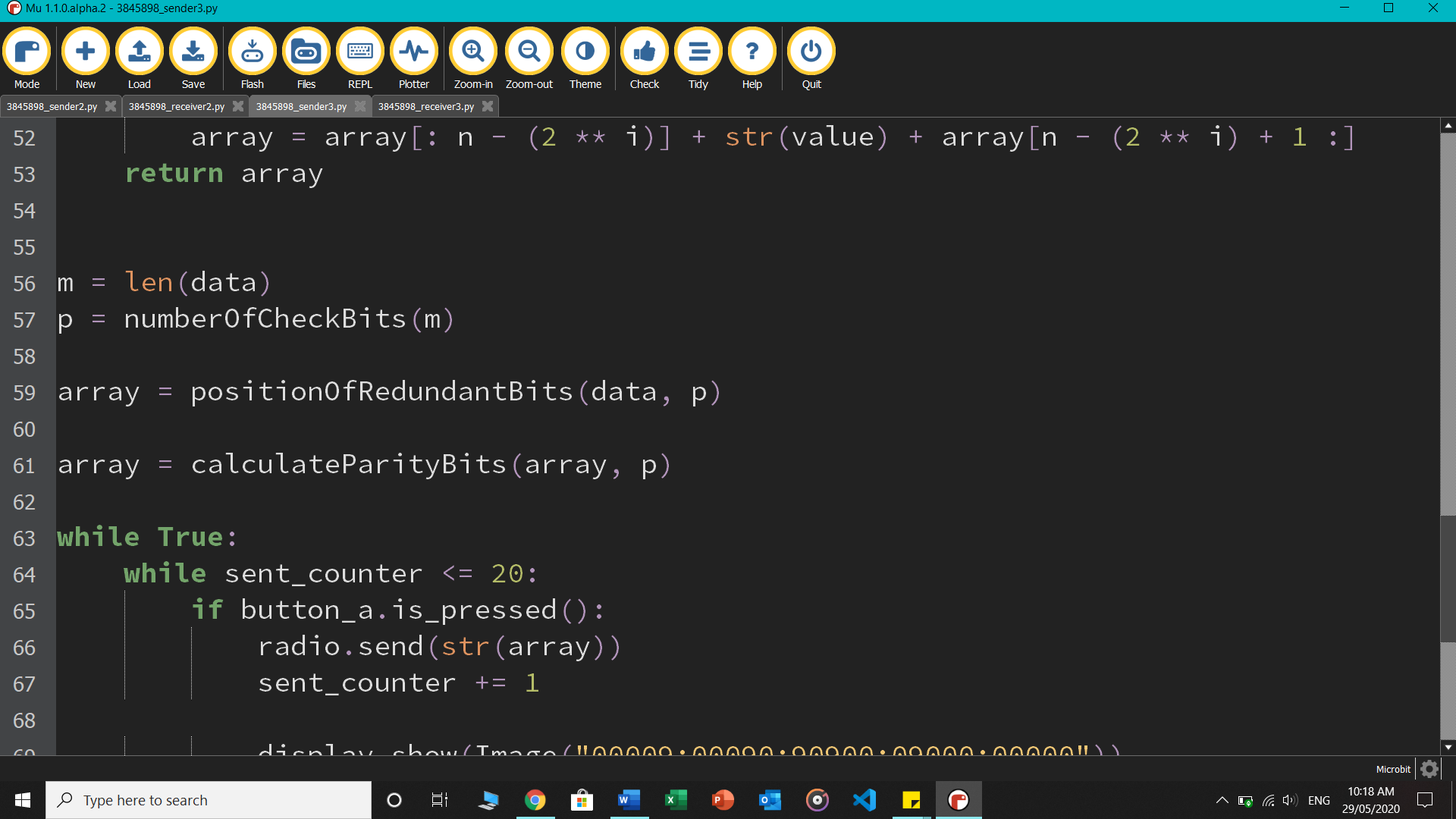
## **Design of the packets**

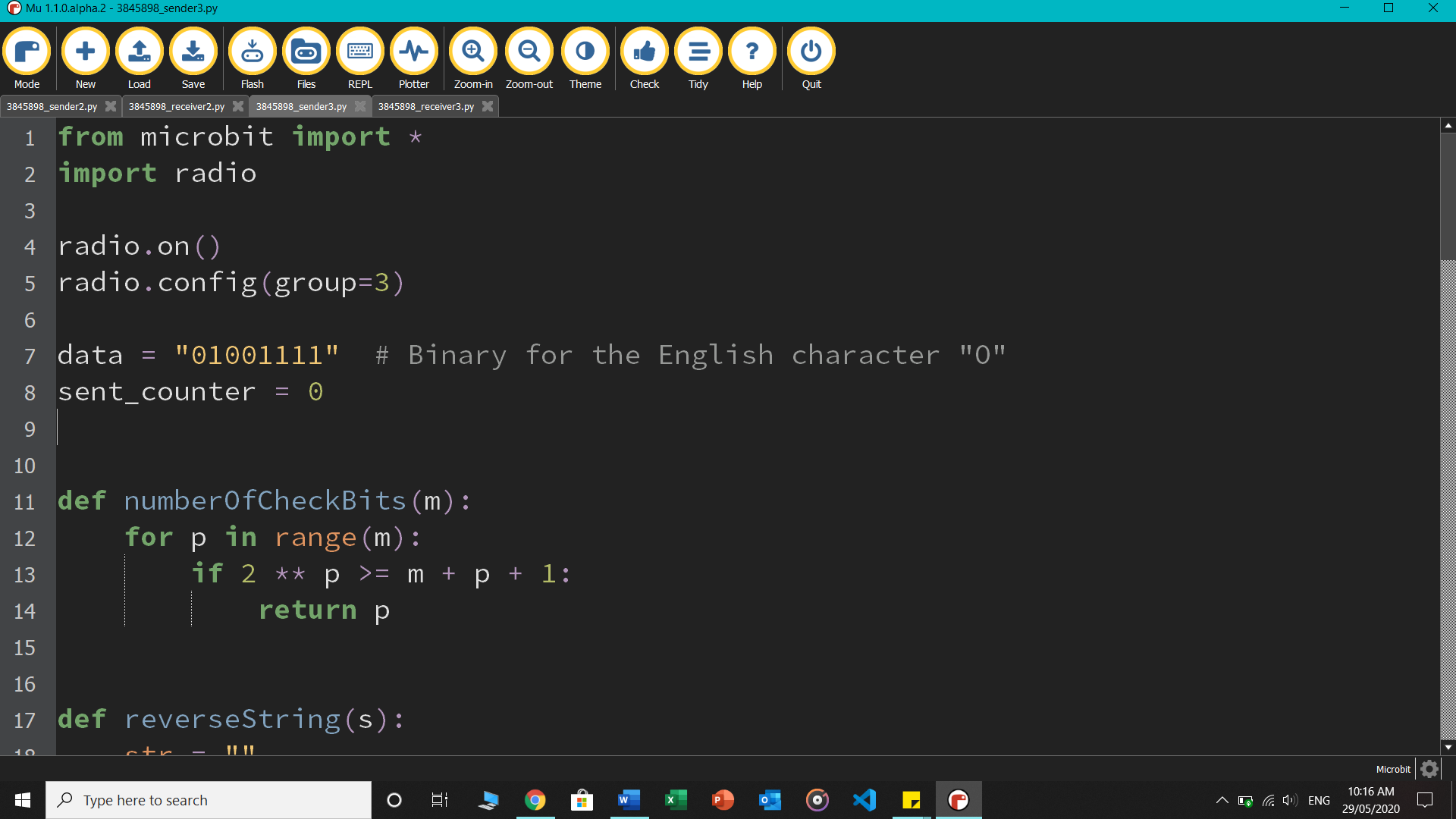
The structure of the packets for this task was different to the other tasks. Since the sender was designed to find, check and alter the parity bits of the pre-defined data (explained more below), the message that it sent through consisted of the calculated parity bits and not just the original data. For simplicity, the sender and receiver address was removed from the packet structure as this task was mainly focused on error detection.

|  |
| --- |
| Data |

**Codes used to designate various states or action**

* The binary number for the English character "O" was devised: “0100 1111”, and used as the required data
* [SENDER] Following variables are defined: m = length of the data; p = no. of check bits required; then using the formula *2p ≥ m + p + 1*, total number of check bits required are calculated and returned by the *numberOfCheckBits(m)* function

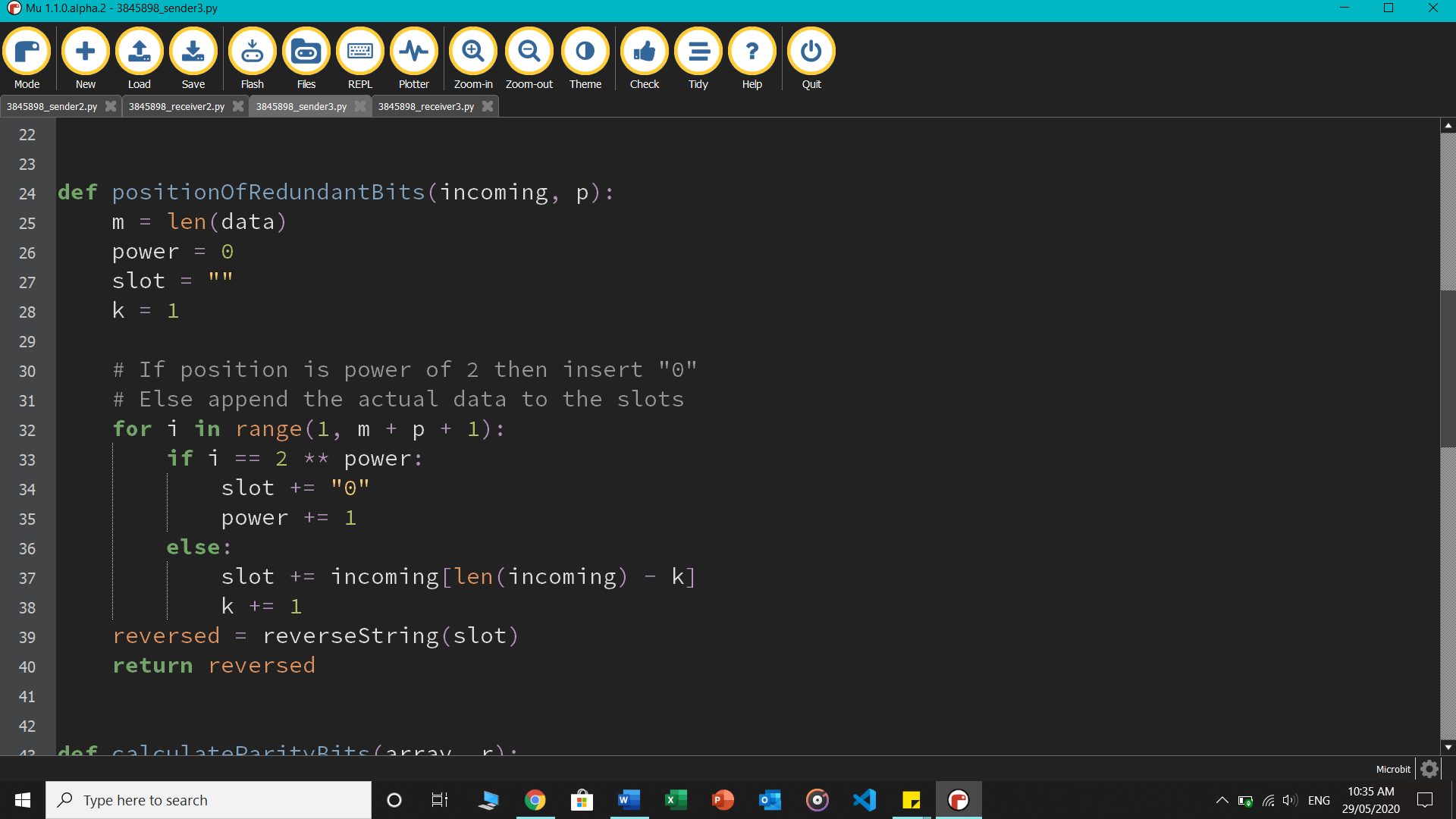


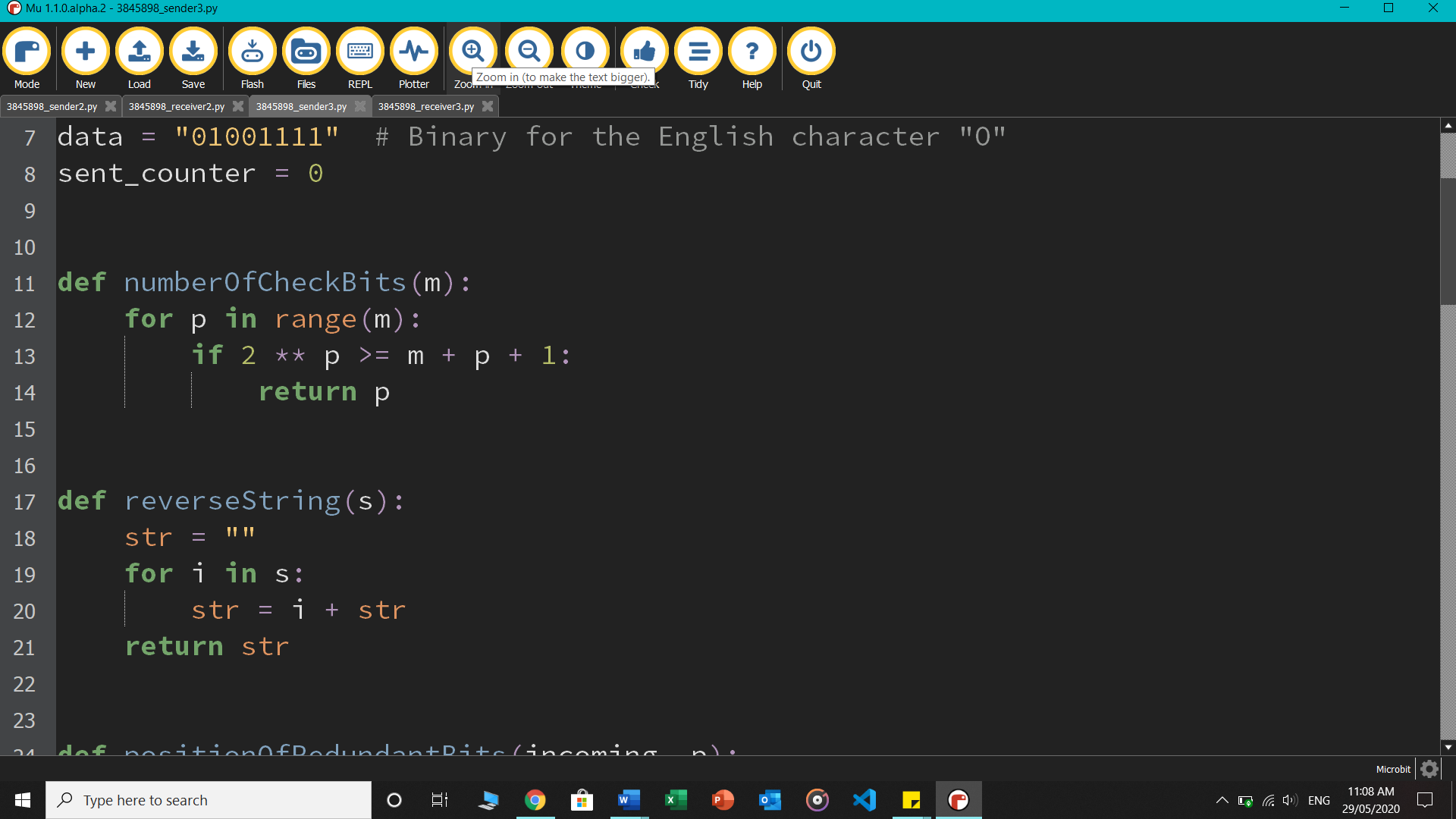


* So far we know that P1, P2, P3 and P4 will be required for the error correction of this data. Now we try to construct the second row of this table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Parity | D8 | D7 | D6 | D5 | P4 | D4 | D3 | D2 | P3 | D1 | P2 | P1 |
| Redundant bits | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |

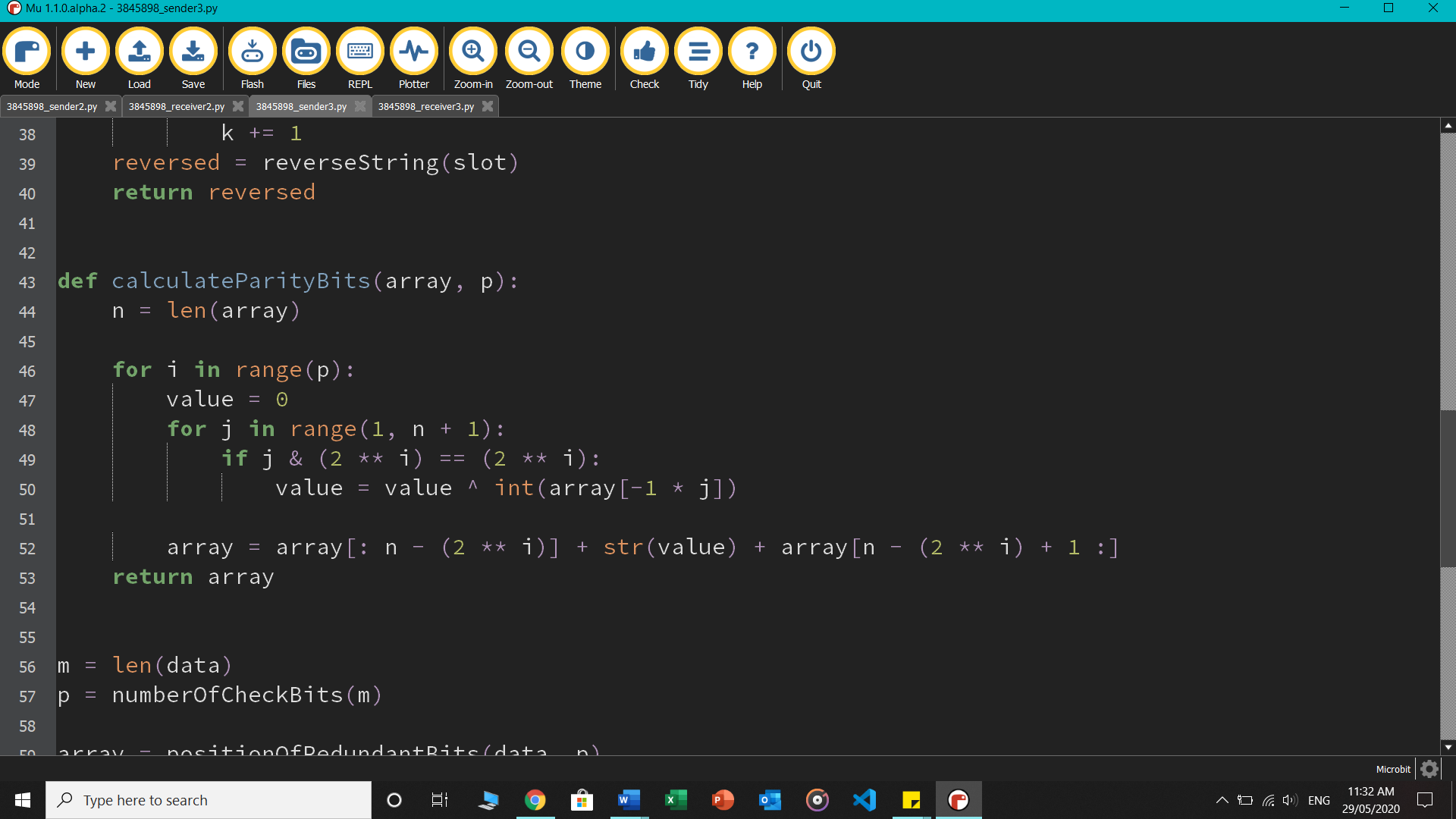
A for loop runs from 1 to 12, where if ‘i’ equals 20, 21, 22 or 24, it adds 0 to the result string named *slot*, or else appends it with the actual data value from the back. Lastly, when the loop finishes, the positions in the *slot* string are reversed using the *reverseString()* function so it matches the table above



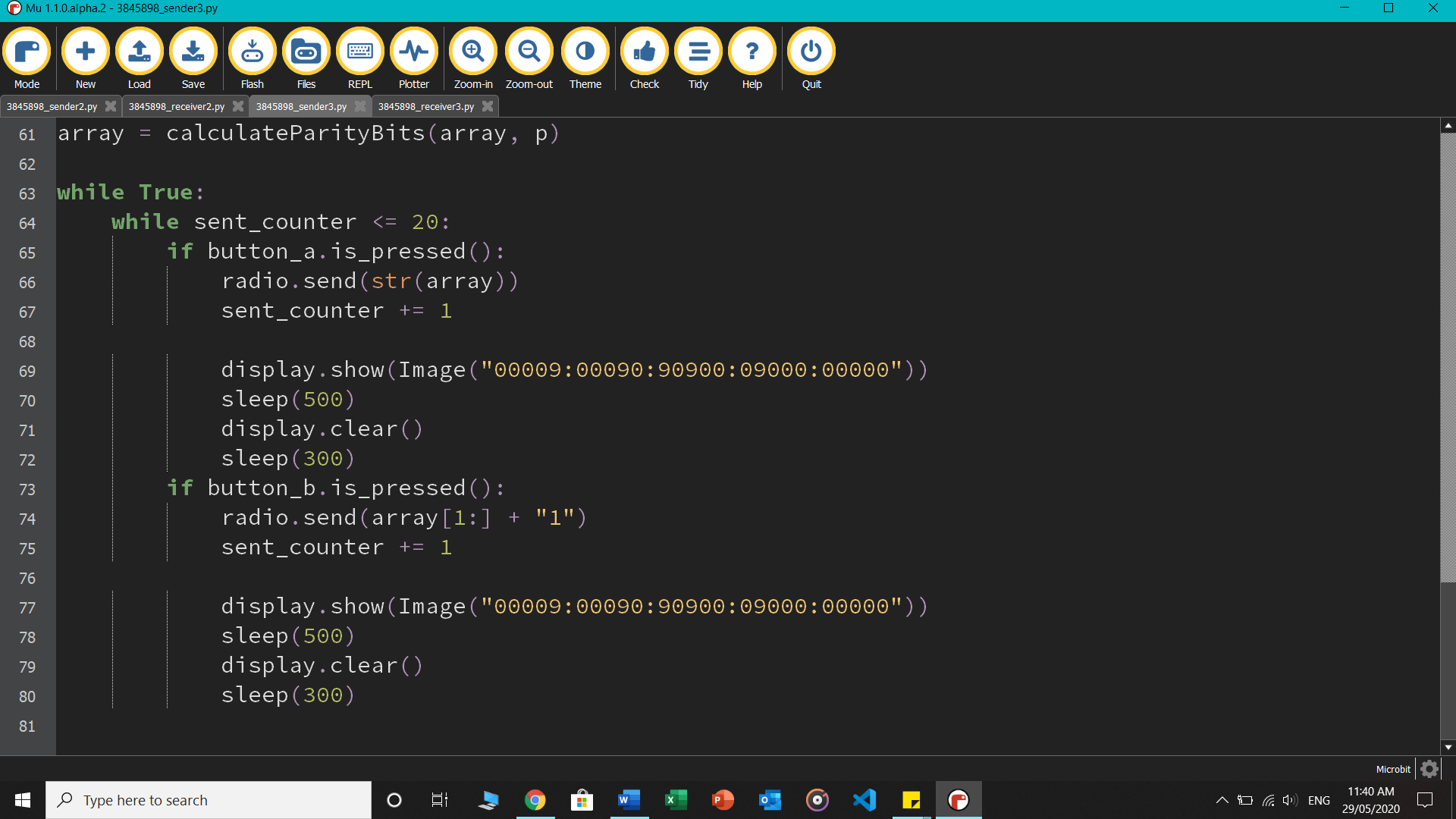


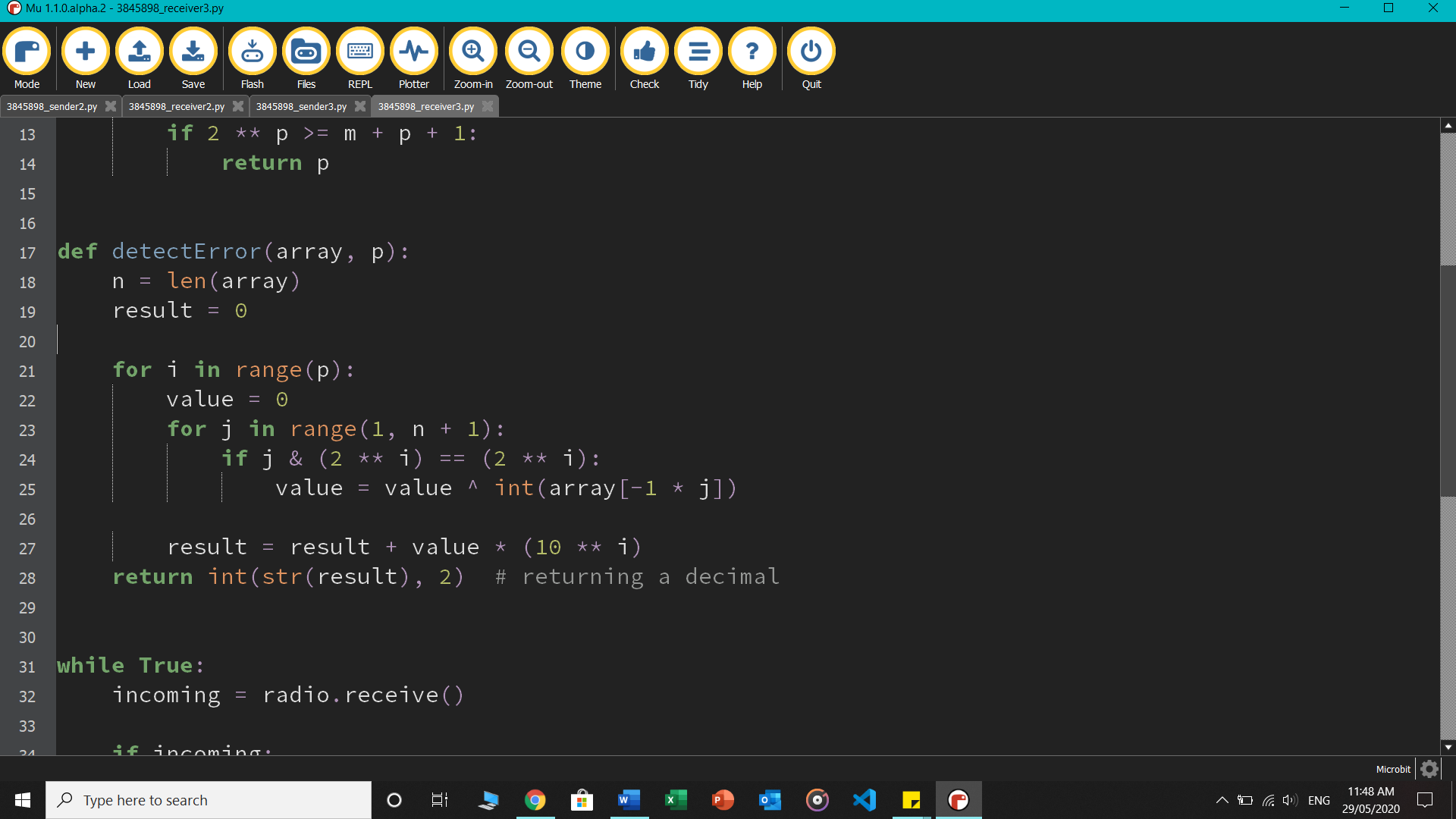
* The reversed string *slot* was stored in array, which is now passed to the *calculateParityBits()* function so that it can complete the rest of the table below by iterating over each parity P1, P2, P3 and P4 [total p=4], and applying error checking rules for each:

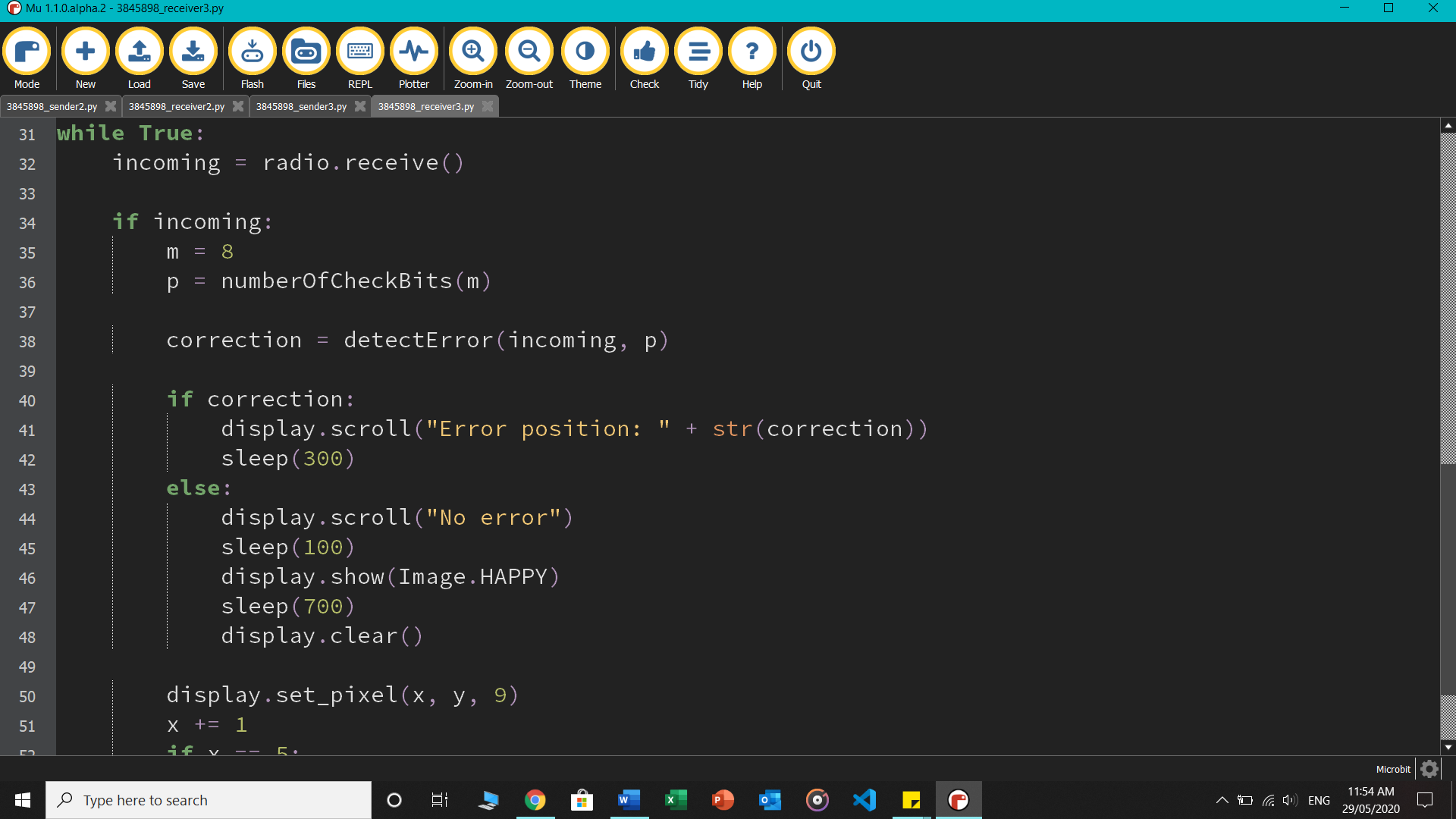
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |
| Parity | D8 | D7 | D6 | D5 | P4 | D4 | D3 | D2 | P3 | D1 | P2 | P1 |
| Redundant bits | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| P1 |  | 1 |  | 0 |  | 1 |  | 1 |  | 0 |  | 0 |
| P2 | 0 | 1 |  | 0 | 0 |  | 1 | 1 |  | 1 | 0 |  |
| P3 | 0 |  |  |  |  | 1 | 1 | 1 | 0 |  |  |  |
| P4 | 0 | 0 | 0 | 0 | 0 |  |  |  |  |  |  |  |



* The new array values containing the calculated parity bits are returned and stored in *array,* which is then sent by pressing on pressing button A. The same packet constituted of the array value is sent 20 times, or in other words, 20 similar packets are sent in a loop. An error can be stimulated in the packet by pressing button B, where the first bit position is omitted and the last value is forced to be “1” so the modified packet now contains an error as it is sent to the receiver.



* [RECEIVER] As soon as the packet containing the array of parity bits arrives it is passed to the *detectError()* function where each parity value is examined by a nested for loop and resulting integer *result* is converted to decimal before returning 
* Finally, if correction is true and the function detects an error then the error position is scrolled on the microbit, otherwise the else condition runs and happy face is shown.



# **TASK 4 – Address Resolution Protocol Implementation**

**Link to the demo videos**

* Task 1 demo: <https://rmiteduau-my.sharepoint.com/:v:/g/personal/s3845898_student_rmit_edu_au/ERf4YHsKARBLq_lVDSP9AjIBL1Zsl8iCKjNS8ag9RLQMDg?e=Fggjgl>
* Task 2 demo: <https://rmiteduau-my.sharepoint.com/:v:/g/personal/s3845898_student_rmit_edu_au/EQk4ubDuiVNNuqU08H-GZoMBqO67WNnfIBrA19022koRwA?e=b3AAbS>
* Task 3 demo: <https://rmiteduau-my.sharepoint.com/:v:/g/personal/s3845898_student_rmit_edu_au/ESW2aleegcVHvg6YhE4pBKkBdF7Y4mcvogrYJfDqfwWSPQ?e=y4G476>

**REFERENCES**

**[1]** "Introduction - Networking with the micro:bit", *Microbit.nominetresearch.uk*, 2020. [Online]. Available: https://microbit.nominetresearch.uk/networking-book-online-python/. [Accessed: 11- May- 2020].

**[2]** "Stop and Wait ARQ - GeeksforGeeks", *GeeksforGeeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/stop-and-wait-arq/. [Accessed: 20- May- 2020].

**[3]** "A Protocol Using Go-Back-N", *Tutorialspoint.com*, 2020. [Online]. Available: https://www.tutorialspoint.com/a-protocol-using-go-back-n. [Accessed: 21- May- 2020].

**[4]** "C++ Program To Implement Go Back N", *Mycodecamp.blogspot.com*, 2020. [Online]. Available: https://mycodecamp.blogspot.com/2019/03/c-program-to-implement-go-back-n.html?m=1. [Accessed: 21- May- 2020].

**[5]** "Hamming Code implementation in Python - GeeksforGeeks", *GeeksforGeeks*, 2020. [Online]. Available: https://www.geeksforgeeks.org/hamming-code-implementation-in-python/. [Accessed: 22- May- 2020].

**[6]** W. do?, I. Vazquez-Abrams and A. Martelli, "What does the caret operator (^) in Python do?", *Stack Overflow*, 2020. [Online]. Available: https://stackoverflow.com/questions/2451386/what-does-the-caret-operator-in-python-do. [Accessed: 22- May- 2020].

**[7]** "8tiqa/go-back-n-udp", *GitHub*, 2020. [Online]. Available: https://github.com/8tiqa/go-back-n-udp. [Accessed: 25- May- 2020].

**[8]** "ASCII Binary Character Table", *RAILWOUND 2012*, 2020. [Online]. Available: http://railwound.weebly.com/home/ascii-binary-character-table. [Accessed: 29- May- 2020].